>>> Final Project for Computer Networks (CNT 4004) <<<

Lily Tang, Christopher Greenland

Introduction

In this network programming project, the goal was to design and implement a file transfer program utilizing UDP, while giving competitive performance and reliability to comparable TCP based solutions. Toward that end, this experimental protocol combines elements of traditional selective repeat protocols, and Network Block Transfer (NETBLT) protocol [6] [7]. It further modernizes the techniques found in these protocols, mainly through the use of POSIX threads, to allow previously discrete and serialized protocol states to be performed in parallel, greatly increasing performance. Implementation and testing has shown that this protocol provides reliable file transfer, in approximately half the transfer time of a TCP-based solution.

Benchmarks and Goals

As per the project requirements, our goals and benchmarks are as follows:

1) Successfully transfer a file between two machines on USF WiFi (local-to-server) and two local hosts (local-to-local) using UDP socket programming.
2) Deliver reliable transportation for files up to 100MB with up to 2% packet loss.
3) Obtain a runtime efficiency within at least 10% of the benchmark TCP protocol's transfer time in situations of both packet loss and no packet loss.

Basic Protocol Operation

The proposed UDP protocol behaves as follows:

Using a request-response paradigm, the server (see Figure 1) waits for a connection from the client. A connection begins with the receipt of a file size, which causes the server to enter the receive state, where it is ready to receive data packets. Each data packet begins with a header containing a sequence number and data length value. The server uses the file size to determine the maximum expected sequence number, based on a fixed packet size. As data packets arrive, an array is updated to track which packets have been received. In the test implementation, lost packets are handled in parallel with the receive state using a POSIX threads, which continuously checks for missing sequences numbers, and transmits resend requests to the client. Alternatively, if threads are not used, the server can transition to a discrete check state through a data timeout, where it checks for missing packets, requests retransmission, and then returns to the check state upon receiving more data. A wait-for-resend state with a timeout accounts for the loss of resend request packets. In either threaded or unthreaded implementation, the incoming data packets are written to the file, using their sequence number as an index to account for out-of-order packet receipt. When the server determines that all sequences numbers have been received, the server enters the done state, and transmits a file acknowledgment packet to inform the client that is has received the entire file. In an ideal implementation, the server waits for a reply to its ack from the client, and retransmits if necessary.

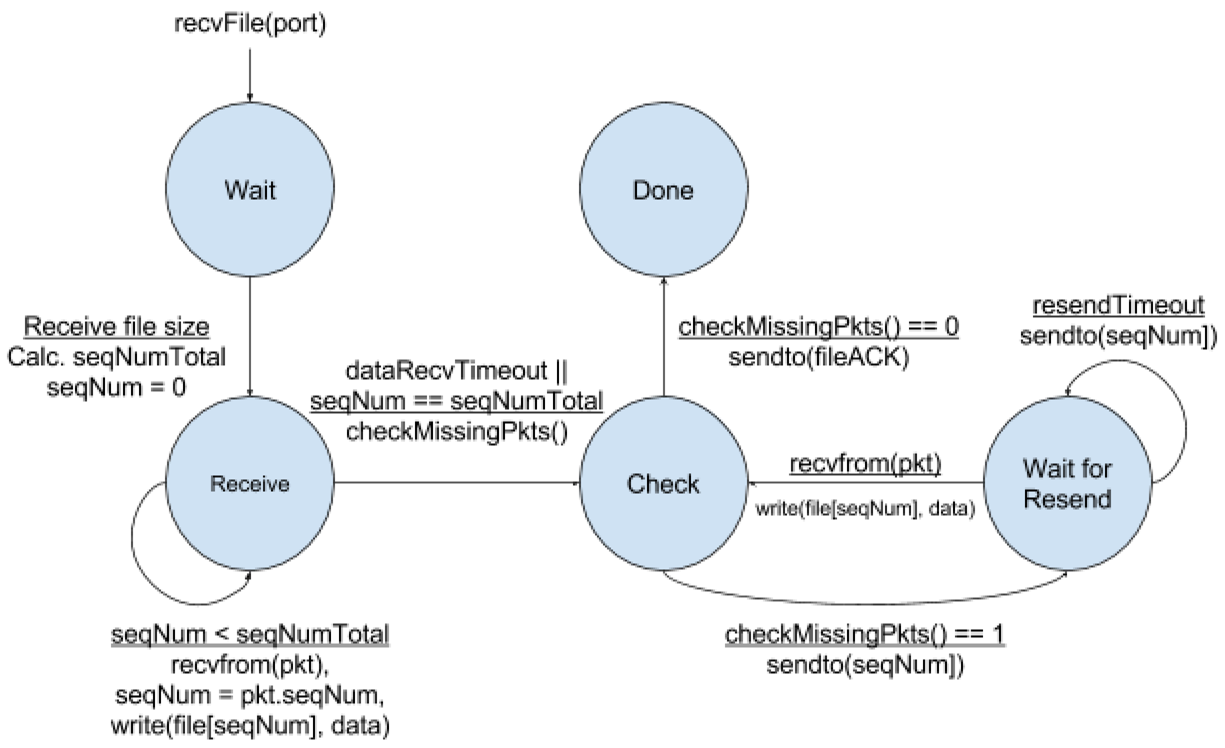

Figure 1 - Unthreaded finite state machine for the server.

At the other end of the connection, the client (Figure 2) begins a transfer by determining the file size, and sending this information to the server using the file size packet. It then enters the send state, where the file is transmitted to the server in an initial series of data packets. Each packet is assigned a unique sequence number in case of loss or reordering, as well as a data size value, since the last packet may vary in size. After the initial transfer, the client must resend any missing packets requested by the server. As with the server, the test implementation of the client uses a POSIX thread to optimize performance. This thread processes retransmit requests from the server, effectively combining the send and resend states. In implementations where threads are not used, the client can transition to a resend state after the initial file send, where it waits for resend request packets, and responds by sending the packet with the requested sequence number again (selective repeat). This continues as a loop until receiving a file acknowledgment from the server, indicating that all sequence numbers have been received, whereupon it enters the done state, and ideally sends a done signal to the server, before closing the program.

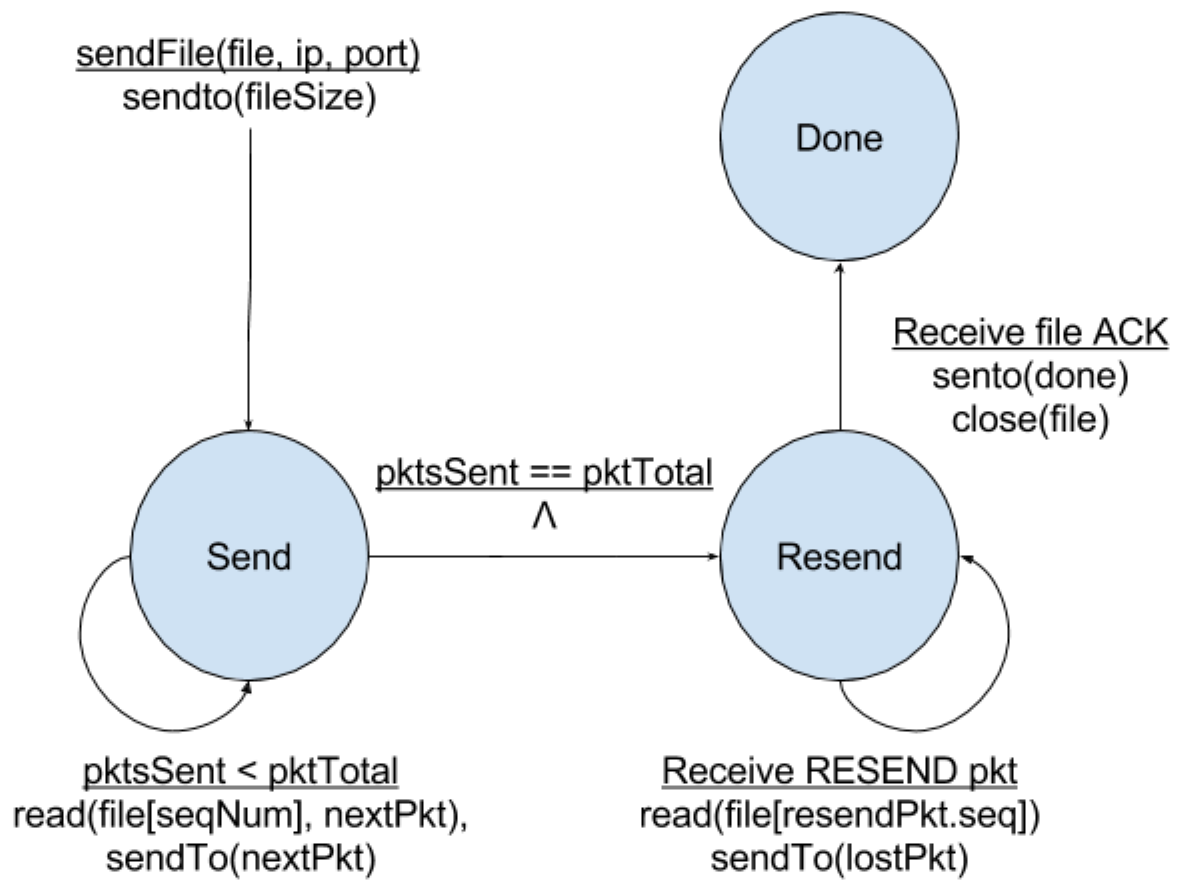

Figure 2 - Unthreaded finite state machine for the client.

Protocol Header and Packet Format

File size packet:

32 bits
File Size (int)

Data packet:

32 bits	32 bits	≤ 1400 bytes
Sequence Number	Data Length	Data…

Resend packet:

32 bits
Sequence Number

Client and Server ACK packet:

32 bits
-1

Design Choices

Many aspects of the project requirement influenced our design choice. Namely, mistakes we made from past attempts at implementing another protocol (Go Back N), as well as runtime execution techniques and efficient data structures used in C greatly drove the design choices made.

i. Lessons from Past Iterations (Go Back N)

Past UDP protocols we've tested include the Go Back N protocol, but this had significant runtime delays during packet loss simulation tests due to the retransmission period and discard packets. In order to keep the packets in sequential order, Go Back N required the client to re-send multiple packets in a single sliding window whenever a packet loss occurred anywhere except the last frame of the window. Our results showed that our Go Back N protocol was on average 300% slower than TCP with a 2% packet loss. From this attempt, we decided to start again using more sophisticated protocols, NETBLT and Selective Repeat, the latter of which handled retransmissions in a more efficient manner. We also learned to focus our new design to implement parallel threads in order to prevent the slow, inefficient transfer that Go Back N offered.

ii. Final Implementation Design

To make up for the failures of our GBN attempt, the final design we implemented used aspects of Selective Repeat for efficient retransmission. With selective repeat, the sender sends a number of frames specified by the window size without needing to wait for individual ACKs from the receiver [1]. Instead, the receiver is notified as to which frames was missing, and only those frames are retransmitted. This is a direct improvement from GBN, which requires every frame from that point to be sent again [2].

Similar inspiration was also derived by studying the network block transfer (NETBLT) protocol. Aspects such as sending large “blasts” of sequential packets for low overhead, using resend requests instead of acks and timeouts to account for packet loss, and indexing packets into buffers by sequence numbers to account for out-of-order receipt.

iii. Runtime Efficiency Optimization

Programming related design aspects were also used to optimize speed. These efficiency techniques include implementing POSIX threads, mmap, and arrays.

A unique aspect of our protocol design was the use of POSIX threads to greatly improve optimization. Thanks to Selective Repeat, the sender no longer has to wait for a response from the receiver before it can continue to blast out packets. Because of this, we can speed up the file transfer by spinning off a separate thread to handle retransmissions while another task (blasting packets) is running in parallel.

One library function we also used is mmap, which is defined as a “POSIX-compliant Unix system call that maps files or devices into memory” [3]. Mmap is preferred over the traditional file I/O handling library because mmap() can be shared between processes/threads, and is therefore more efficient in our protocol. In addition, mmap provides a multitude of benefits. First, because memory for mmap is allocated by the kernel, it is always aligned. Second, because the memory area may be shared with the kernel block cache, there is less copying involved [4].

Another optimization technique is to use an array where the index of the array indicates the sequence number, and the value denotes a boolean value (1 or 0) indicating whether or not that particular packet has been transmitted yet. Arrays are very useful because the packet sequence numbers are in sequential order using natural numbers just like array indexes. In addition, accessing elements in array is independent of the array size, giving it a runtime of O(1) [5]. As you can see, there are many reasons behind the design choices made, with a large part of our algorithm focused on improving the speed of reliable UDP to match that of TCP.

iv. Tradeoffs and Choices

One major tradeoff of our design is the trade-off between congestion control/fairness versus efficiency and speed. In this project, we sacrificed congestion control by never backing off when the network is busy. Instead, it will continue to blast packets regardless of what others are doing in order to send its file as quickly as possible. This makes our protocol a very selfish design meant to do one thing and one thing only. However, this design also allows our protocol to quickly and reliably get the job done in order to match TCP speeds.

In addition, another tradeoff we considered was efficiency versus ease of implementation. In a pure NETBLT protocol, NETBLT sends one request with a list of packets, which is a little more efficient. Instead, our protocol sends one request per missing packet like SR, which is easier to implement. Because the efficiency difference is perceived to be small, we decided to stay with the SR implementation of packet resend.

Analysis and Future enhancements

In the current test implementation, the window size is not restricted or recalculated for congestion or flow control; this is to maximize performance and simplify implementation. However, future implementations can easily add TCP-like sliding window congestion control, to provide more fairness to other connections at the expense of performance. Overall, the proposed protocol performed well in our experimental tests (Table 1), performing on average 53.3% faster than TCP's file transfer time.

Because this implementation sends the whole file at once, file size is limited to the maximum value of the integer sequence numbers multiplied by the data payload size of each packet. However, this limitation can easily be overcome through the implementation buffers and buffer numbers, as found in the NETBLT protocol. Sending the file one or more buffers at a time, each containing many packets.

Additional improvements could include client-server negotiation of packet size, send rate, and other parameters; as well as improvements to timeouts and message transmission algorithms.

Step #	Disc.	Time 1	Time 2	Time 3	Mean	% of TCP Benchmark
1	TCP Local	29.842	30.608	25.017	28.489	100%
2	UDP No Loss Local	14.457	14.415	14.251	14.374	50%
3	UDP 2% Loss Local	14.743	14.413	14.389	14.515	51%
4	TCP WiFi	39.631	43.040	42.427	41.699	100%
5	UDP WiFi	19.794	19.605	19.607	19.607	47%

Table 1 - Preliminary testing of protocol implementation (prior to official project evaluations).

References

[1] https://en.wikipedia.org/wiki/Selective_Repeat_ARQ
[2] https://en.wikipedia.org/wiki/Go-Back-N_ARQ
[3] https://en.wikipedia.org/wiki/Mmap
[4] https://stackoverflow.com/questions/9817233/why-mmap-is-faster-than-sequential-io/-9818286-#9818286
[5] http://bigocheatsheet.com/
[6] https://tools.ietf.org/html/rfc969
[7] https://tools.ietf.org/html/rfc998